May 2016

An Empirical Study on The Impact of C++ Lambdas And Programmer Experience

Phillip Merlin Uesbeck
University of Nevada, Las Vegas

AN EMPIRICAL STUDY ON THE IMPACT OF C++ LAMBDAS AND PROGRAMMER EXPERIENCE

by

Phillip Merlin Uesbeck

Bachelor of Science – Applied Computer Science
Universität Duisburg-Essen
2014

A thesis submitted in partial fulfillment of
the requirements for the

Master of Science in Computer Science

Department of Computer Science
Howard R. Hughes College of Engineering
The Graduate College

University of Nevada, Las Vegas
May 2016

UNLV | GRADUATE COLLEGE

Thesis Approval

The Graduate College
The University of Nevada, Las Vegas

April 15, 2016

This thesis prepared by

Phillip Merlin Uesbeck

entitled

An Empirical Study on The Impact of C++ Lambdas And Programmer Experience

is approved in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science
Department of Computer Science

Andreas Stefik, Ph.D.
Examination Committee Chair

Jan Pedersen, Ph.D.
Examination Committee Member

Kazem Taghva, Ph.D.
Examination Committee Member

Matthew Bernacki, Ph.D.
Graduate College Faculty Representative

Kathryn Hausbeck Korgan, Ph.D.
Graduate College Interim Dean

Abstract

Lambda functions have become prevalent in mainstream programming languages, as they are increasingly introduced to widely used object oriented programming languages such as Java and C++. Some in the scientific literature argue that this feature increases programmer productivity, ease of reading, and makes parallel programming easier. Others are less convinced, citing concerns that the use of lambdas makes debugging harder. This thesis describes the design, execution and results of an experiment to test the impact of using lambda functions compared to the iterator design pattern for iteration tasks, as a first step in evaluating these claims. The approach is a randomized controlled trial, which focuses on the percentage of tasks completed, number of compiler errors, the percentage of time to fix such errors, and the amount of time it takes to complete programming tasks correctly. The overall goal is to investigate, if lambda functions have an impact on the ability of developers to complete tasks, or the amount of time taken to complete them. Additionally, it is tested if developers introduce more errors while using lambda functions and if fixing errors takes them more time. Lastly, the impact of experience level on productivity is evaluated by comparing the performance of participants from different levels of experience with one-another. Participants were assigned one of five levels based on their progress in the computer science major. The five levels were freshman, sophomore, junior, senior and professional. The professional level was assigned to individuals out of school with 5 or more years of professional experience.

Results show that the impact of using lambdas, as opposed to iterators, on the number of tasks completed was significant. The impact on time to completion, number of errors introduced during the experiment and times spent fixing errors were also found to be significant. The analysis of the difference between the different levels of experience also shows a significant difference. The percentage of time spent on fixing compilation errors was 56.37% for the lambda group while it was 44.2% for the control group with 3.5% of the variance being explained by the group difference. 45.7% of the variance in the sample was explained by the difference between the level of education. Therefore, this study suggests that earlier findings that student results are comparable with the results of professional developers are to be considered carefully.

Acknowledgements

"I want to express my gratitude to my advisor Prof. Andreas Stefik of the University of Nevada, Las Vegas for continuously supporting my research and giving me the opportunity to research this topic to begin with. Additionally, I want to thank Prof. Petersen for his help during the design of the experiment and the writing of this thesis and Prof. Taghva and Prof. Bernacki for agreeing to be part of my committee.

Further, I want to thank Dr. Stefan Hanenberg of the University of Duisburg-Essen for introducing me to this topic and taking his time for many, many endless meetings to teach me what I needed to know to design my first experiment.

I also want to thank my parents unwavering support and my girlfriend for being patient with me during the experiment for—and the writing of—this thesis.

Finally, I want to thank the participants of the experiment for taking the time to contribute to this research and the rest of the team of Prof. Stefik's lab for their help and support during the design and preliminary testing of the experiment."

Phillip Merlin Uesbeck

University of Nevada, Las Vegas

May 2016

Table of Contents

List of Tables

List of Figures

List of Code Samples

Chapter 1

Introduction

Programming languages are widely spread over many different domains and are used for a number of different use-cases. They are the basis on which our modern information society is built with technology as far ranging as controlling traffic lights, nuclear facilities, space shuttles, the internet, and the smartphone in everyone's pocket.

Examples of widely-used programming languages are Java and C++, as they are deployed in a wide variety of commercial settings. These languages are frequently changed with the hope to improve them and their usability, effecting developers that use them every day. Along with these modifications come many effects on the infrastructure built around the languages. Textbooks need to be changed and developers have to be trained. This can impact development schedules, which has tangible development costs.

Along with trying to avoid excessive costs for programming language features that give no benefit to developers using a certain language, research on programming language features can also increase the general understanding of what design choices can improve productivity with programming languages. Research in software engineering has found that the highest cost factors in software development are development time and maintenance [Boe88]. This makes it reasonable to assume that improvements in developer productivity with the chosen programming language might reduce overall software cost as writing and maintaining code should cost less time. While research focuses on the technical performance of programming languages, such as compilation and execution speed [ATCL$^+$98, FKR$^+$00, VRCG$^+$99, BS96] and tools for debugging and development [KM09, HCIB04], the usability of the programming language itself is only investigated rarely [Han10]. In fact, a survey by Kaijanaho found that only 22 randomized controlled trials on programming language features were published between 1976 and 2011 [Kai15].

To conduct research on the impact of programming language features on developer productivity, randomized controlled trials can be used, as they provide researchers with a lot of control over the experimental environment. This enables researchers to study the interaction between the variables of the experiment closely, which, in turn, enables the researchers to get a detailed understanding of the impact of changes.

No randomized controlled trials on the topic of lambda functions could be found concerning the design and usefulness of lambda functions as they were recently introduced in Java 8 and C++11. So, this thesis will describe a randomized controlled trial to investigate the impact of lambda functions on programmer productivity to increase the body of evidence on this topic. More specifically, the experiment in this thesis compares the human productivity difference between lambda functions and iterators for iteration tasks. Productivity is measured in time to achieve a complete program, in which a program is defined as complete if it fulfills a set of guidelines measured by unit tests.

As the participants for the experiment, students from the computer science program of the University Nevada, Las Vegas were recruited. A number of professional developers with more than 5 years of experience also volunteered. The diversity of participants, in regards to their standing in the academic pipeline, allowed the experiment to investigate claims made in the literature that the difference between the performance of students and professionals is small and that results from similar studies with only students as participants are generalizable to professional developers [SMJ15, HRW00].

The results of this study show that there is a slight, statistically significant, negative impact on the amount of time the participants took to write a program correctly when using lambdas compared to iterators. This was also supported when looking at other variables such as the amount of errors the participants encountered during the experiment and the time they spent in a non-compiling state.

The outcome of the experiment also suggests that there is a significant difference between the different stages of education and professional developers when it comes to productivity. The professional developers successfully solved more of the tasks and took less time to develop correct programs than students. This finding calls the external validity of previous experimental results with students into question.

This thesis is based on an experiment also described in a paper that is going to be published at ICSE 2016 under the same title and, as such, there will be many similarities between the two works.

The rest of this paper is structured as follows: In chapter 2, the reader will be provided with background information on the design of lambda functions arguments for and against them. Then in chapter 3, information on similar work in the field will be given. More information on the design of the experiment can be found in chapter 4. Results of the experiment will be described in chapter 5 and a discussion of the results can be found in chapter 6. Threats to the validity of the experiment will be discussed in chapter 7 and future research ideas will be described in chapter 8. The thesis will finally conclude in chapter 9.

Chapter 2

Background

This chapter will focus on what lambdas are and in which context they are investigated in this thesis. First, the theoretical basis will be described and then the implementation in modern languages will be explained.

2.1 Lambdas

2.1.1 Theoretical Basis

Lambda expressions, or functions, are a programming language construct based on Alonzo Church's lambda calculus from 1932 [Chu32], which was used to model theoretical computability. The following will be largely based on the descriptions of lambda calculus from [Pie02].
A valid λ-expression can either be:

1. a variable, such as x.
2. an abstraction, such as $\lambda x.t$, where t is another λ-term.
3. an application, such as $(x\ y)$, where x and y are λ-terms.

A variable, in a modern programming language sense, would be a common variable or parameter.

An abstraction is like a function, method, or procedure definition in a modern programming language. The λ-abstraction can contain other λ-terms inside of it. A difference between the modern language function definition and a λ-abstraction is that the abstraction has no name, can only have one parameter, and is completely untyped. Later variations of the original λ-calculus have added rules for types and other constructs to expand the possibilities of what can be modelled in λ-calculus. A λ-abstraction with two parameters could be build by having a second abstraction inside the first one like this:

$$\lambda x.\lambda y.t$$

where t is another λ-term. Abstractions translate to first-order functions in contemporary programming languages, which are function definitions that can be contained in and contain other function definitions.

Applications represent the act of calling a function in a programming language and passing a parameter to that function. The parameter that is being applied can be another λ-term, such as variables and abstractions or value. Applications can not be passed since they would be evaluated first, unless lazy evaluation is used. This can also be found in contemporary implementations of lambdas in that lambda function definitions can be treated like values and be passed as parameters to methods or other lambda functions. It is important to note that the λ-applications would only apply one parameter at a time, as this is limited by the number of parameters that λ-abstractions can have. To apply two variables to a function would look like this:

$$((\lambda x.\lambda y.t\ a)\ b)$$

where a and b are λ-terms. In this, a would first be applied to $\lambda x.\lambda y.t$ by replacing all free instances of x in $\lambda y.t$ with a and removing the "$\lambda x.$" part. Then b would be applied to $\lambda y.t$ by replacing all instances of y in t with b and removing the "$\lambda y.$" part. One should remember that t, a, and b in this example can always also be λ-abstractions. These rules are the basis of most functional programming languages such as Lisp, Scheme, Haskell, and JavaScript.

2.1.2 Lambdas in Object-Oriented Programming Languages

In the context of object-oriented programming languages, lambda functions usually refer to anonymous functions that can be treated like values and also be defined inside of other functions. As they are treated like values, they can also be passed to other functions as parameters. Another interesting feature of lambda functions is that they can refer to names in their surrounding scope.

While the object-oriented language Smalltalk had lambda functions (called blocks) early on, many more wide-spread object-oriented languages like Java, C++ and C# did not. C++ introduced lambda functions in C++11 after having a similar feature: function pointers. Java had another feature similar to lambdas in its anonymous inner classes before introducing lambda functions in Java 8.

Anonymous inner classes, however, are implementations of interfaces or subclasses of other classes and can require the implementation of multiple methods. They can have their own members like other classes and they can access the members of the enclosing class. Anonymous inner classes can also access local variables from their enclosing scope, but only if they are declared as $final$ or must be effectively final. Anonymous inner classes introduce issues with shadowing, however, which comes from the class having its own scope in which new declarations shadow declarations from the surrounding scope. Lambdas, which are lexically scoped in Java 8, do not introduce a new level of scoping and cause and error when an existing variable would be redeclared. One could argue that the syntax of anonymous inner classes is more verbose than that of lambda functions, which is not to say that either is easier or harder to work with.

The difference between function pointers in C++ and the newly introduced lambda functions in C++11, is that the function pointers do not allow to capture the surrounding scope of the application of the function unless references are passed as parameters. Additionally, C++ without lambda functions does not allow for the definition of functions within other functions.

When the feature was introduced to C++11 and Java 8, in both cases it was claimed to be more intuitive to use than similar features that can perform mostly the same functionality.[1] According to Willcock et al., lambda functions are "syntactic sugar for defining function object classes" [WJG+06]. They go on to say that lambda functions must be "significantly less verbose than defining explicit function objects, not significantly more verbose than explicit loops and clearer and more intuitive than explicit loops." The C++ designers continue on that "A lambda expression has aspects of a function (it performs an action) and an object (it has a state). The primary aim is for lambda expressions to serve as 'actions' for STL[2] algorithms (the way function objects have traditionally been used) and similar callback mechanisms."

2.1.3 Definition of Lambda in C++ and Java

A technical definition of the C++11 lambda syntax is defined as:

$$[cl]\ (P_1\ p_1, P_2\ p_2, \ldots, P_n\ p_n)\ \rightarrow\ t\{s_1; s_2; \ldots; s_n\}$$

The *cl* in the first brackets represents the capture list and can either be a number of variables to capture from the scope, either as a reference or as a value, or be a & or a = to capture all variables in the scope as reference or as value, respectively. It can also be left blank to not capture any variables. The element right after the capture list—$(P_1\ p_1, P_2\ p_2, \ldots, P_n\ p_n)$—represents the list of parameter names and their types. The t denotes the return type of the function and can be any kind of C++ type. If no return type is explicitly stated, then the return type is inferred. The arrow is optional if no return type to the function is given. $\{s_1; s_2; \ldots; s_n\}$ is a sequence of statements and the last statement of the sequence can be a return statement. A concrete example in C++ syntax can be seen in code sample 1.

Code sample 1 shows an example without captured scope. The output of this code would be "6". The example also shows how the syntax without explicit return type can be used, which also means that the arrow is not necessary. Additionally, this example shows the function type that was introduced with C++11. It takes the form *function*$< r(P_1, P_2, \ldots, P_n) >$ whereas the r is the return type and $(P_1, P_2, \ldots, P_n)$ defines the parameter types. It defines the type of the variable holding the lambda function and can be substituted for the *auto* keyword to use inference.

The Java programming language chose the syntax as follows:

$$(p_1, p_2, \ldots, p_n) \rightarrow \{s_1; s_2; \ldots; s_n\};$$

where p_i is a parameter in form of a variable name and $\{s_1; s_2; \ldots; s_n\}$ is a list of statements. The last statement can be a return statement. A specific example of this syntax can be seen in code sample 2. In

[1]For a more in-depth discussion of these claims see section 2.2.

[2]Standard Template Library

```
function<int (int, int)> func = [] (int p, int q) {
  int s = q+p;
  return s-p;
};
int a = 5;
int b = 6;
cout << func(a,b) << "\n";
```

Code Sample 1: C++ lambda function example.

the example, `Operation` is an interface with a method taking two integers as parameters and returning an integer and can be seen in code sample 3.

```
Operation o = (p,q) -> {
  int s = q+p;
  return s-p;
};
int a = 5;
int b = 6;
System.out.println(o.doOperation(a, b));
```

Code Sample 2: Java lambda function example.

```
interface Operation {
  public int doOperation(int p, int q);
}
```

Code Sample 3: Operation interface.

The lambda functions in both of these languages are usable to, for example, provide callbacks for events, such as for button presses and network messages. They can also be used to create a function that can be passed to methods that operate on data structures. An example for this could be a method that iterates over a list of numbers and applies a function to each of the numbers in turn. The function can be provided as a parameter to the method by the programmer, giving the method more reusability as different functions can be supplied, which was one of the motivations for adding the feature [Ben13, WJG$^+$06, Sam06].

2.2 Arguments For and Against Lambdas

To get a better view of why lambda function implementation was deemed important enough by Oracle and the C++ ISO committee to invest the development time into, a literature review was conducted. The aim was to find arguments for and against lambda functions in marketing materials and other non-peer-reviewed publications as there was little to be found in the scientific literature.

2.2.1 Arguments for Lambdas

During the JavaOne technical keynote in 2013, Mark Reinhold, Oracle's Chief Architect of the Java Platform Group, stated that lambda expressions are "the single largest upgrade to the programming model ever - larger even than generics." Brian Goetz, Oracle Java Language Architect, followed this up with "Programming well is about finding the right abstractions. We want the code we write to look like the problem statements it's trying to solve, so we can look at it and immediately know it's correct. Java has always given us good tools for abstracting over data types. I wanted to do better in abstracting over patterns of behavior - that's where lambda comes in… Lambdas are a nicer syntax, but they are also something deeper. It's not just a compiler generating inner classes for you - it uses the invokedynamic feature to get more compact and higher performance code." The summary of their claims was that "lambda brings 3 weapons to Java - syntax, performance and abstraction… plus parallelism" [Ben13]. It should be mentioned at this point, that a study on syntax showed that novice programmers had about as many problems with Java's syntax as with a randomly generated language [SS13].

In their document about the implementation of lambda functions in C++, Willcock et al. state that they think that key libraries of C++ weren't used as often as they should be, because "The lack of a syntactically light-weight way to define simple function objects is a hindrance to the effective use of several generic algorithms in the Standard Library" [WJG$^+$06]. Samko argues in 2006 that "Many algorithms in the C++ Standard Library require the user to pass a predicate or any other functional object, and yet there is usually no simple way to construct such a predicate in place. Instead one is required to leave the current scope and declare a class outside of the function, breaking an important rule of declaring names as close as possible to the first use. This shows that lambda functions would add a great [sic] to the expressive power and ease of use of the C++ Standard Library [Sam06]."

Harvey advocates for an extension of the block-based Scratch language to include procedures as arguments to other procedures which would bring lambda functionality to the language. He states that "these have proven to be a powerful capability… They are useful, for example, as an alternative to recursion or to looping with index variables when the programmer wants to process all of the elements of a list in a uniform way" and goes on to say that "by taking key ideas, such as procedures as first class data, from the Scheme language, we can add only a few features to Scratch and still make it powerful enough to support a serious introductory computer science curriculum [HM10]."

Lastly, a common argument amongst supporters seems to be that a number of other languages have implemented the feature already: "Many programming languages offer support for defining local unnamed functions on-the-fly inside a function or an expression. These languages include Java, with its inner classes; C# 3.0; Python ECMAScript; and practically all functional programming languages." Further, claims go that "anonymous functions within other functions, with access to local variables, are an important feature in many programming languages" [JFC08].

2.2.2 Arguments Against Lambdas

The inclusion of lambda functions might be argued against by pointing out that adding new ways to solve tasks that were already solvable in different other ways does not necessarily improve a language. They confuse programmers as to which way is the 'better' one in a specific case. This might lead to unnecessary discussions or other wastes of time in a project.

Bjarne Stroustrup, the original designer and implementer of C++, states in a FAQ on his web-page that the "Primary use for a lambda is to specify a simple action to be performed by some function ... A lambda expression can access local variables in the scope in which it is used... Some consider this 'really neat!'; others see it as a way to write dangerously obscure code. IMO, both are right [Str14]." This brings up the point that lambda functions could lead to hard-to-read and complex code.

In a blog post called "The Dark Side of Lambda Expressions in Java", the author argues that adding lambdas to Java can create a bigger divide between the Java code and how the JVM byte-code was formed [Wei14]. This can possibly lead to a greater difficulty in understanding stack traces, which can lead to harder-to-debug programs.

Chapter 3

Related Work

As argued before in section 2, studies on the effect of programming language constructs on human developers have been rare. This section will show the results of a number of literature surveys on the topic as well as a number of randomized controlled trials that have influenced the design of this experiment.

3.1 Literature Surveys

To find studies on empirical evidence about lambda functions, several literature surveys have been consulted. Studies by Kitchenham et al. and Zhang et al. do not mention any research on lambda functions or related topics [KPBB^{+}09, ZBT11]. A literature survey on the existence of empirical programming language construct studies does not mention any relevant studies either [Kai15]. A systematic review of different academic workshops about the impact of programming language design on developers by Stefik et al., which coded the quality of evidence in the reviewed papers, found no papers about lambdas either [SHM^{+}14]. A similar review of the ICFP and OOPSLA venues conducted internally for the publication of this experiment did not find relevant studies with sufficient evidence standard.

3.2 Repository Mining

Another methodology to research the usage of programming language features is the use of repository mining. That is, analyzing large amounts of code and the change of code over versions using the data found in version control systems—also known as source control management tools—such as SVN[1], Git[2], or Mercurial[3]. This is most easily done by analyzing large numbers of open-source projects as they are easily available. Such studies can especially give a good picture of how the adoption of newly introduced features is progressing. One example for such a study can be seen in Parnin et al.'s research on the adoption of generic types in

[1]https://subversion.apache.org/
[2]https://git-scm.com/
[3]https://www.mercurial-scm.org/

the Java language [PBMH11], in which it is shown that this relatively newly introduced feature did not meet much adoption in open source projects. Other, similar, studies can be seen in [RPFD14] and [NF15]. While these types of studies can provide researchers and language designers with great information about the impact of programming language constructs on the development process and source code, results can only be attained after a change was made to the language and the developers of the projects under analysis had a chance to adopt the new feature. This means this methodology is can only be used after worldwide deployment, which makes it less useful when trying to determine if a feature should be implemented.

3.3 Research on Programming Language Constructs

Experiments on Syntax: In an attempt to learn more about the effect of programming language syntax the researchers Stefik et al. [SS13] ran 4 experiments. In an experiment with 196 students, they investigated which programming language syntax constructs are most intuitive. They found that familiarity with the C++ programming language correlated with increased rating for how intuitive C++-style programming language syntax is, suggesting that a rating of intuitiveness from an experienced programmer might be biased. Non-programmers rated words higher that are common in English and describe the purpose in a literal way. One example for this was the finding that "repeat" was rated higher than "for" for iteration tasks.

Additionally the researchers analyzed how accurately programming novices could learn different programming languages in a limited time-frame. The accuracy of the programs written by the students was rated. This showed that static type annotations were problematic for novices and that braces, parentheses, and semicolons in the languages tended to cause errors. It was also found that accuracy was higher if the language under test was using syntax that was rated as more intuitive.

Studies focusing more on errors were performed by Denny et al. [DLRTH11, DLRT12], who found that the most common errors are "Cannot resolve identifier", "Type mismatch", and "Missing ;". Petersen et al. [PSV15] found that 86% of the errors in C code they analyzed were linker errors and undeclared identifiers, as well as missing semicolons and other syntax mistakes. A similar analysis with Python code found that 73% of the errors encountered where syntax errors. An analysis of Java code showed that more than 70% of the errors found were some sort of error related to syntax. Another study on errors in student code was conducted by Altadmri and Brown [AB15] and found that other than mismatched parentheses and brackets, semantic and type errors are the most common kinds of errors. While the results of these studies do not completely agree on what type of error is most common, they show that students are impacted heavily by the syntax and semantics of a language.

Static Versus Dynamic Type Systems: An experiment by Gannon [Gan77] investigated how many errors programmers make when using typed languages compared to untyped languages. He found that statically typed programs were less error-prone than their untyped counterparts.

A study about how type declarations are used in Groovy by Souza and Figueiredo [SF14] found that types are more often used in public methods than in private ones, that types are less often used in tests and scripts and that they are less often used in files that are changed often over the course of a project. Further, they found that developers with experience in developing with dynamically typed programming languages tend to use fewer types in their programming than developers who only used statically typed languages.

A randomized controlled trial by Kleinschmager et al. [KHR+12] found that static type systems help development time over dynamic type systems in unfamiliar code. The experiment also found that there is an advantage when using statically typed languages over dynamically typed languages when fixing type errors. The study was later replicated with code completion in IDE's by Petersen, Hanenberg and Robbes [PHR14]. Another study found in 2014 that static type systems have a positive impact on development time when using documented APIs [EHRS14].

A study using the Dart programming language by Samuel Spiza found that the use of type names helps in the use of APIs, even if the type checker is disabled [SH14]. The study also found that this is only true as long as the type annotations stay correct and that a wrong annotation can cost participants extra time.

Finally, a paper on the use of generic types in Java utilized a randomized controlled trial to test if there is a difference between using generic types and raw types when using unfamiliar APIs. The researchers found that generic types seem to give an advantage, but extending class structures such as the strategy pattern was significantly harder when using generics than when using raw types [HH13]

Other Studies: Rossbach et al. [RHW10] found in a study about transactional programming that it was perceived as harder by the participants when it was compared to coarse-grained locks and slightly easier when it was compared to fine-grained locks. Analysis of the errors made by students found that participants of the experiment introduced fewer errors when they used transactional memory, as opposed to locks.

A series of experiments by Daly et al. [DBM+96] found that a program with a class structures with an inheritance depth of 3 were quicker to maintain than an identical program that was flattened, the effect did not hold for inheritance structures with a depth of 5 compared to a flat program. A replication by Cartwright [Car98] found the opposite to be true. A further study of the topic more focussed on understanding than changing by Prechelt et al. [PUPT03] found that maintenance was faster for programs with less inheritance depth. Prechelt et al. go on to say that their findings in combination with the previous results suggest, that inheritance depth might not have an influence on maintenance effort and that the number of methods that participants have to understand is a better predictor.

3.4 Programmer Experience

Siegmund et al. [SKL+14] evaluated ways that programmer experience is measured in controlled experiments. They found that the most used measurement was the number of years programmers had spent programming. Others used the level of education and self-estimation as well as other means. They also found that many of the publications covered in their literature review did not control for the level of experience of programmers or did not specify how it was done. In an experiment with questionnaires and code-evaluation tasks, Siegmund et al. found that self-evaluation questions from the questionnaires seemed to correlate well with the number of correct answers in the code-evaluation tasks. Especially self-evaluation in respect to students comparing themselves to classmates and in respect to their experience with logical programming languages were found to be good estimates for programming experience in a student population.

Research by Crk et al. [CKS15] found that programming expertise could be measured using EEG. In their study, participants of different levels of a four year college curriculum had to solve programming tasks. The class levels correlated with measured brain waves. Interestingly, the experiment found that students with class level 2[4] performed the best when it came to correctness while class level 4 had the second best correctness. The researchers suggest that this result might be explained with the heavy focus on programming skill and the Java programming language around the point in the curriculum students are in classified as level 2. Fritz et al. [FBM+14] have found that perceived task difficulty for developers can be predicted using eye-trackers and other psycho-physiological measures.

On the topic of comparison between students and professionals, a study by Höst et al. [HRW00], investigated the difference between the performance of students and professionals when rating the importance of factors that contribute to the lead-time of a software engineering project. The study had 26 students and 18 professionals. The researchers conclude that there is no big difference between students and professionals in that sort of rating task and state that this probably holds true for masters students of computer science in their last year.

Another study by Salman et al. [SMJ15] investigated the difference in code quality, measured by a number of different code metrics, between students and professionals. The quality metrics did not include if the code runs or works correctly. The programming experiment involved using test-last and test-first development. Seventeen graduate students were compared to 24 professionals. The study found that there was a difference in quality between students and professional developers when it comes to test-last development, but that there was no difference in the test-first development tasks. The researchers harbor doubt that the small effects they found might be due to differences in tasks the participants worked on. They conclude that there is no difference between students and professionals as long as the topic is new to both groups and that the test-first development differences show that experience with a topic makes a significant difference.

Research by Youngs [You74] used an experiment with 30 students and 12 professionals to investigate which kinds of errors are common when the participants work on a number of programming tasks and how the amount of errors changes while programmers are working on the task. Programming languages used in this experiment were Algol, Basic, Cobol, Fortran, and PL/1. Youngs found that novices committed more errors on average than their professional counterparts. The average amount of errors between the two groups was the same for the first run of the program.

[4] Approximately year 2.

A study by Wiedenbeck [Wie85] found that in an experiment, in which students who were in a programming class on Fortran and professionals using Fortran were asked to evaluate if code samples shown to them on a screen were correct or not, that professionals were significantly more accurate and faster in their decisions than the students. In a second experiment described in the same paper, the same populations had to evaluate if a description was fitting a given code sample. Differences between the two groups were once again significant in favor of the professional programmers.

Chapter 4

Experiment

The experiment described in this thesis recruited participants to solve a number of programming tasks. Each participant was assigned to one of two groups. The first group had to use the lambda function language feature to solve the tasks while the second group had to use the iterator design pattern instead. This second group functioned as the control group since the iterator design pattern is a commonly used way to do iterations in programming languages—such as C++—and could be called the "normal" way to go about solving such tasks. The focus on iteration in this experiment was chosen because of claims that iteration tasks are easier using the new feature [HM10]. Further claims—such as that the use of parallel programming is easier in certain cases[Ben13]—were not investigated due to the limited nature of a randomized controlled trial. C++ was chosen for the experiment as it is the language that is most commonly learned at UNLV, where the study was planned to recruit most heavily.

4.1 Hypotheses

This experiment aimed to find evidence on the impact of lambda expressions in iterating a data structure compared to iterators. It aims to answer the following null hypotheses:

$H0_1$: There is no impact on the number of tasks developers can correctly solve using lambda expressions compared to using iterators in C++.

$H0_2$: There is no impact on the time it takes developers to correctly complete programming tasks using lambda expressions compared to using iterators in C++.

$H0_3$: There is no impact on the number of compiler errors developers have when they complete programming tasks using lambda expressions compared to using iterators in C++.

$H0_4$: There is no impact on the percentage of time developers spend in a non-compiling state when using lambda expressions compared to using iterators in C++.

$H0_5$: Experience level, defined by the position an individual has within an academic pipeline or professional status, has no impact on developer performance under any condition.

We can not reject a null hypothesis unless the difference between the two groups of data is significant. The statistical difference level often chosen in programming language research is 5%. This is the typical amount of statistical significance used in other sciences such as psychology and sociology [LFH10, 34].

If the groups are significantly different then the null hypothesis is rejected. This means that the opposite of the null hypothesis is true with a confidence in the results of at least 95%, depending on the p-value.

4.2 Participants

The student-participants for this study were recruited in the computer science department of the University of Nevada, Las Vegas. We recruited students from the classes CS202, CS302, CS326 as well as CS460 and CS473. In exchange for the effort, the students were awarded extra credit. The amount of extra credit was based on what the individual instructors were willing to agree to and ranged from 1 to 2%.

Professional developers were also recruited by asking for volunteers on a mailing list that are interested in functional programming as well as on twitter. Interested individuals were encouraged to get into contact with the proctor to arrange for a time to do the experiment.

Overall, 54 participants were found. Of these, 10 were freshmen, 8 sophomores, 17 juniors and 7 seniors as well as 12 professionals. The categorization of which year the student participants were in was made based on which class they were enrolled in at the time of the experiment.

The experiment was conducted at the beginning of a semester and freshmen were enrolled in their second programming class, CS202, or Computer Science II, which concerns itself, according to the official enrollment information, with "Data structures and algorithms for manipulating linked lists. String and file processing. Recursion. Software engineering, structured programming and testing, especially larger programs". A requirement for this class is to take CS135, also known as Computer Science I, which is "Problem-solving methods and algorithm development in a high-level programming language. Program design, coding, debugging, and documentation using techniques of good programming style. Program development in a powerful operating environment." It seems reasonable to assume that students of this level have a basic understanding of programming.

The sophomores were enrolled in CS302, Data Structures, which is described as "Introduction to sequential and linked structures. File access including sequential, indexed sequential and other file organizations. Internal structures including stacks, queues, trees, and graphs. Algorithms for implementing and manipulating structured objects. Big-O-notation." The requirement to take this class is to have finished CS202 and a math class. The students in that class did not cover the topic of iterators or lambdas in class at the time of the experiment.

Juniors were enrolled in CS326, Programming Languages. The description of this class states that the following is covered: "Design, evaluation and implementation of programming languages. Includes data types and data abstraction, sequence control and procedural abstraction, parameter passing techniques, scope rules, referencing environments and run-time storage management. Study and evaluation of a number of current programming languages." More importantly, students taking this class must have taken and completed CS302, have completed CS219 or another class on computer organization and have advanced standing.

Seniors were enrolled in either CS460, Compiler Construction, which is described to cover "Current methods in the design and implementation of compilers. Construction of the components of an actual compiler as a term project.", and requires the completion of CS326, Programming Languages, and CS456, Automata and Formal Languages. Or they were enrolled in CS473, which is a practical class on software engineering and requires another class on software engineering to be taken before it. In both cases, it is reasonable to assume that the students are close to the end of their computer science curriculum and must have gained considerable amounts of knowledge in a variety of computer science topics, including programming.

4.3 Study Protocol

Before the experiment was conducted, a checklist was created, so that the experiment was conducted in the same way for every participant. This checklist was used on every student-participant in the study.

The professional participants were distributed throughout the US and Europe and to include them in the experiment the steps were changed slightly as the experiment was conducted using Skype[1].

4.3.1 Recruiting

To ensure the same conditions for the participants, they were treated as similarly as possible from the start. The experiment advertisement was read in the five classes during normal class hours while copies of it were handed around the classroom. Then, interested students had and opportunity to ask questions about the time commitment, the reward in form of extra credit and the room the study would happen at, as well as inquiries about the possible dates and times when they could come in. Questions about the methodology and content of the experiment were not answered to avoid the good subject effect and compensatory rivalry [NM08, CC05]. After and during the questions, interested students were able to register for a time-slot on a piece of paper that was handed around.

It should be noted that every student was able to earn the extra credit even if they did not participate in the experiment by sending the proctor an essay about a computer science topic of their choice. The length of the essay had to be a page per possible credit percent earnable as extra credit. This was implemented to

[1]http://www.skype.com/en/

allow students to earn the extra credit without having to participate in the experiment if they chose not to. The instructors of the participants received a list from the proctor of the experiment about who earned the credit.

For the professional developers, an e-mail was sent by professor Stefik on a functional programming mailing list. Professor Stefik also sent out a tweet about the experiment on his personal Twitter account, which was forwarded by other prominent persons in the programming community to ensure that as many people as possible would read and reply to it. Interested developers answering either the tweet or the e-mail by sending a direct e-mail to the proctor would be sent the same advertisement information as the students in the classes had received.

4.3.2 Room setup

To prepare the room for every time slot prospective participants had registered for, the needed computers were prepared. The setup of the workstations was chosen in a way that made it hard for participants to be distracted by the screens of other participants, by creating distance between the participants' workstations.

Furthermore, the workstations had to be set up to provide the correct environment to the participants. This was done by making sure that VirtualBox[2] was installed on each computer, as well as making sure that the correct virtual machine was prepared on it. Were these conditions met, the proctor made sure to reset the working environment inside the virtual machine to ensure no information from previous participants was interfering with new participants. Finally, the proctor made sure the virtual machine was maximized on the screen and that the task folder for each work station was opened for the right group. Then, the proctor distributed a number of print-outs at each workstation. One print-out was a number of pages of a code sample, depending on the group the workstation was assigned to. The second sheet was distributed to inform the participants about the process and their rights. This was an important document as it had to be signed, else the participants could not take part in the experiment. Another print-out was the study protocol which was to be read by, and to, the participants. The last print-out was a questionnaire about the previous experience of the participants.

Since the professional participants from all over the world were not able to attend the lab time-slots due to their geographic distribution, the setup of the experiment had to be done differently for them. To ensure smooth running of the experiment in the time the participants had allotted to partake, the proctor helped them prepare the necessary software environment beforehand communicating via e-mail and instant messaging. They were instructed to download VirtualBox, or a similar tool like VMWare, and import the `.ova` file which contained the necessary virtual machine image for the experiment. They were also asked to test if the virtual machine would boot up without error, but not to look at any of the content of the virtual machine. They were furthermore asked to install Skype, if they hadn't already, and to register an

[2]https://www.virtualbox.org/wiki/Download

account if needed. The proctor prepared a Google Drive[3] folder for each participant. The folder contained the documents for the specific group of the professional participant, like the ones given to the student participants as print-outs.

4.3.3 Operation

The participants were assigned to workstations based on order of arrival. During the time of arrival, the proctor signed off the participants that showed up to make sure they would get their promised extra credit. To ensure as little distraction as possible, the proctor waited for the arrival of all participants before starting the experiment. When all participants were present and seated, the proctor instructed the participants to read and understand the informed consent sheet and if they agreed, to sign it. When all participants had either signed the consent form or left the room, the proctor instructed the participants to read the study protocol, which informs the participants about the specifics of how the experiment was going to work. To ensure the participants read and understood the study protocol, the proctor read the protocol to them and asked to affirm their understanding. The participants were then to study the code samples, which were assigned to each work station, for 5 minutes. After that, they were instructed to start the first task. Participants kept the samples and were allowed to refer back to them at any point.

The professional developers were called on Skype at the agreed-upon time. The screen sharing option was used so that the proctor could see what the participants were doing. Then they received access to the Google Drive folder. They were asked to sign the consent form electronically and e-mail it to the proctor. Then they were read the study protocol and encouraged to follow along. After that, they were also asked to read the code sample for 5 minutes. When the five minutes were up, the participants were asked to start the first task. The proctor watched the participants remotely using the screen sharing function to make sure the participants were following the rules. Especially to prevent them to use copy&paste from the code sample if they were looking at it on their computers instead of printing it out as that would have been an action the student-participants could not take.

Each task had a number of automatic tests to ensure that a task was completed correctly. If a participant felt like the task was done, they could click a button and have the program test their solution, which would either result in a pop-up informing them that they are done or in the output of the compiler and test environment being displayed to them. In case they were not done yet, they were able to make further changes to the program and try again. In case they were done, they were to dismiss the dialog and close the IDE to stop the task and move on to the next one. If there was no further task for them to do, they were done and were asked to fill out the questionnaire that was provided to them. After this they were no longer needed and were asked to leave.

Due to ethical and practical considerations, the design of the experiment took into account that some of the participants might not be able to solve a task in a practical time frame. To address this issue, a

[3]https://www.google.com/drive

maximum time per task was put into place, so that participants which had spent 40 minutes on a single task were asked to move on to the next one. The task was then marked as not completed. This time-frame was decided upon based on data from a pilot-study using the same tasks. The decision to add a maximum time did, however, add an upper limit to task times, which did influence the analysis of the results. One can argue that the limit added a conservative view to the analysis, as waiting for the tasks to be completely done might have increased the differences between the groups even more.

4.4 Tasks

Each participant was asked to solve 4 tasks. Each task was provided as a program with the code inside a single method missing for it to correctly run. The participants' responsibility was to fill in that method to complete the program and finish the task. A description of what the method was supposed to do was provided to the participant as a comment above the method signature. The file they were supposed to work in was opened automatically upon starting the task.

The tasks also often contained other source and header files to provide the participants with more context and enable them to solve the task. For example, a task might have all programming logic contained in its own source file, but its header and a source and header file for a data type that was also used in the logic was provided in the project folder as well. The participants could look at these other files at any time during the task.

The very first task was the same between both the groups, as it was the warm-up task. This first task involved using a simple loop to iterate over a vector object to get the participants used to the environment. This was especially important to get them used to the procedure of starting the task, solving the task and using the tests to confirm that the task was correct by pressing the “Run/Proceed”-button. The second task was the first task specifically for the assigned group.

Task two, three and four were focusing on using the group-specific feature to solve the task. The feature for each group and its use was shown in the code sample that the participants of that group received at the beginning of the experiment. The comments in the tasks asked the participants explicitly to use the feature that was shown in the code sample, so that they would not improvise a solution using other means. The tasks were always provided in the same order.

4.4.1 Warmup Task

This is the first task any participant had to solve. This task's purpose was to get the participants used to the environment and reduce impacts of the environment on the other tasks' measurements. Additionally, the hope was that participants would be able to refresh their knowledge of how to use C++. Thus, every participant had this same task first, independent from which group they were in. In it the requirement is

```
#include "task.h"

using namespace std;

/**
 * Please implement a loop which puts all elements of numbers into the
 * retVal variable.
 */
vector<int> getValues() {
    vector<int> numbers;
    numbers.push_back(1);
    numbers.push_back(5);
    numbers.push_back(65);
    numbers.push_back(21);

    vector<int> retVal;

    //Implement solution here
    // --------

    int index = 0;
    while (index < 4) {
      retVal.push_back(numbers[index]);
      index++;
    }

    // --------

    return retVal;
}
```

Code Sample 4: Warmup Task.

only to use a simple loop. The participants are deliberately informed which content is in the vector they have to use.

A possible solution to this task was to use a counting variable and a while loop, or alternatively a for-loop. Then the participants had to use the `push_back()` method to put the values into the `retVal` vector. A solution as found in the results of the experiment can be seen in code sample 4.

4.4.2 Task 1

This task was the first task in which the participants had to use the given programming construct. In this task the participants had to use the given `marketBasket` object to achieve their task.

For the iterator group, the expectation was that the participants use the iterator object that they could call from the `marketBasket` object using the `begin()` method. For a look at the `marketBasket` object provided to the participants of the group see code sample 5. When the participants acquired the iterator, they could use the `hasNext()` method as the loop condition in a while or other loop and add the price of

```
#include "marketBasket.h"

using namespace std;

void marketBasket::insert(string itemName, float itemPrice) {
    item newItem;
    newItem.name = itemName;
    newItem.price = itemPrice;
    items.push_back(newItem);
}

marketBasket::iterator::iterator(marketBasket *owner) : owner(owner), index(0) {
}

void marketBasket::iterator::next() {
    index++;
}

bool marketBasket::iterator::hasNext() {
    return owner->items.size() - index > 0;
}

item marketBasket::iterator::get() {
    return owner->items[index];
}

marketBasket::iterator marketBasket::begin() {
    return marketBasket::iterator(this);
}
```

Code Sample 5: Iterator group `marketBasket`.

the current item to the `retVal` variable that was provided to them. The last important step was to iterate to the next item using the `next()` method on the iterator, either in the body of their loop or inside the loop header of a for loop. An example of how a participant chose to solve the task can be seen in code sample 6.

The participants of the lambda group were provided with a different `marketBasket` object as can be seen in code sample 7. The participants had to use `iterateOverItems()` by providing it with a function with the return type `void` and an `item` as argument. Examples of how to use lambda functions were provided in the code sample handout. The participants had to adapt the correct argument type, however, to get the task complete. This means they had to understand which part of the `function<void (person)>` function type of the sample code denotes the type of the argument. The type they needed was `function<void (item)>` which was also visible in the `marketBasket` class, as it is the type that `iterateOverItems` expects. A challenge in this task was also to recognize, that the lambda function would be able to use a variable from the scope it was defined in. This was also visible in the code sample. With that knowledge, participants could use the `retVal` variable directly and add the prices to it in the function they defined. A working example from how a participant solved the task can be found in code sample 8.

```
#include "task.h"
#include "marketBasket.h"

using namespace std;

/**
 *  Please write a program that calculates the sum of the prices of
 *  the items using the same technique as seen in the sample code.
 *  Assign the result to retVal.
 **/
float getSum(marketBasket mb) {
    float retVal = 0;
    // Implement solution here
    // --------
    marketBasket::iterator iter = mb.begin();
    while(iter.hasNext()){
        retVal+=iter.get().price;
        iter.next();
    }
    // --------
    return retVal;
}
```

Code Sample 6: Task 1 iterator group.

Another part of the syntax of defining the lambda functions, the `[&]`, which defines how the scope is captured, was deliberately shown in the code sample the same way as it had to be used to prevent the participants from getting confused over this part. It was expected that this experiment would be run on a diversity of different levels of education and thus it was deliberately decided to avoid additional scoping details.

4.4.3 Task 2

This task was the second one in which participants had to use the specific programming language construct. This task asked the participants to find the lowest and the highest value in the given object and to return their sum ($lowest + highest$).

The iterator group had to use their `marketBasket`, which was the same as the one used in task 1 (code sample 5), to solve the task. The participants were provided with two variables of type `float`, with one being assigned to the lowest possible float number and one being assigned to the highest possible float number, to make it simpler for the participants to go about finding the highest and lowest value by comparing with these values.

The iterator group used the same approach in going through the objects values as before. They had to get the iterator with `begin()`, use `hasNext()` as their loop condition and go to the next item using `next()`. For this specific task, they could just use two if-statement in the loops body to compare the value of the

```
#include "marketBasket.h"
using namespace std;

void marketBasket::insert(string itemName, float itemPrice) {
    item newItem;
    newItem.name = itemName;
    newItem.price = itemPrice;
    items.push_back(newItem);
}

void marketBasket::iterateOverItems(function<void (item)> f) {
    for (vector<item>::size_type i = 0; i < items.size(); i++) {
        f(items[i]);
    }
}
```

Code Sample 7: Lambda group `marketBasket`.

current item to the `highest` variable, which was initialized to the lowest float possible, and if the item's price was higher than the current value of `highest`, they could assign the item's price value to be the new value of `highest`. Similarly they could do this with `lowest`. Because of how the tests were set up, the if-statements could not be mutually exclusive. One of the test-cases tested with just one item expected the final value to be this item's value multiplied by two as the result. With mutually exclusive if-statements, the tests would fail. The return statement of the method was given. One solution from the experiment can be seen in code sample 9.

The lambda group, similar to the iterator group, had to use their `marketBasket` class (code sample 7). The solution to this task is very much analogous to the one for the iterator group. The participants just had to create a lambda function as seen in the first task and use the same structure of if-statements as the iterator group. A complete solution of this task can be seen in code sample 10.

4.4.4 Task 3

Task 3 was the last task for each participant and the third in which they had to use the given programming language feature. The task asked the participants to find the items in the `marketBasket` that have a price below 30 and put them into the `under30` vector using the `push_back()` method. **The iterator group** had to use the same approach as in the previous two tasks. The participants would have to get the iterator with the `begin()` method and use the `hasNext()` method as the loop condition. Then, the participants could just use an if-statement to check if the price of the current item is below 30. If that was true, they could use the `push_back()` method to add the item to the `under30` vector. Then they needed to iterate to the next item using `next()`. An example solution for this task can be found in code sample 11.

The lambda group had to create a lambda function of type `function<void (item)>` and declare the function to compare the current item's price with 30 and, if it was 30, use the `push_back()` method on

```
#include "task.h"
#include "marketBasket.h"

using namespace std;

/**
 *  Please write a program that calculates the sum of the prices of
 *  the items using the same technique as seen in the sample code.
 *  Assign the result to retVal.
 **/
float getSum(marketBasket mb) {
    float retVal = 0;
    // Implement solution here
    // --------
    function<void (item)> summer = [&] (item it) {
            retVal += it.price;
    };
    mb.iterateOverItems(summer);
    // --------
    return retVal;
}
```

Code Sample 8: Task 1 lambda group.

`under30` to insert the item into the given variable. Then the `iterateOverItems()` had to be called with the defined lambda function. An example of this can be seen in code sample 12.

4.5 Experiment Environment and Data Recording

To make the digital part of the experimental environment as similar as possible between different computers and operating systems, it was decided to use a virtual machine. This way, it was possible to conduct the experiment on a number of different systems and still have largely the same environment for all participants. This helped especially with the participants that were not able to physically visit the lab in which the experiment was conducted in and had to use their own computers to take part in the experiment.

Ubuntu was chosen as the operating system of the virtual machine, because it is an easy to set up Linux system that gives participants a more familiar feeling to Windows and Mac OSX systems. Also, the intended development environment for the experiment was known to run well on Ubuntu.

The integrated development environment (IDE) of choice was a modified Eclipse installation. It was decided to use a very simplified system without any supporting features for software development, such as code completion and in-line error display, to give all of the participants the same experience. This way, participants with experience in using IDEs, or more specifically the Eclipse IDE, would not be faster than other participants because they know keyboard shortcuts for the IDE's features that others didn't know or weren't as familiar with.

```
#include <float.h>

#include "task.h"
#include "marketBasket.h"

using namespace std;


/**
 *  Please write a program that calculates
 *  the sum of the lowest and highest item using
 *  code similar to the sample code.
 *  FLT_MIN is the minimum value for a float.
 *  FLT_MAX is the maximum value for a float.
 **/
float getLowestHighestSum(marketBasket mb) {
    float highest = FLT_MIN;
    float lowest = FLT_MAX;
    // Implement solution here
    // --------
    marketBasket::iterator iter = mb.begin();
    while(iter.hasNext()) {
        if(iter.get().price > highest)
            highest = iter.get().price;
        if(iter.get().price < lowest)
            lowest = iter.get().price;
        iter.next();
     }

    // --------
    return highest + lowest;
}
```

Code Sample 9: Task 2 iterator group.

This resulted in the IDE being stripped down to its most basic functions. It only had a very basic text editor with the usual text editing functionality, such as typical writing and editing as is common in other visual text editing tools, saving the file, undoing changes and redoing changes. Features like code completion or automatic error highlighting were not available in the IDE. The IDE also had a project view, which lists all relevant files to the project in a narrow window on the left-hand side of the screen. This list was modified so that irrelevant files would not show up, such as `build.xml` or test files. This was done to not distract the participants from focusing solely on programming. The only visible files were the source code and header files that were important for solving the task.

Most of the buttons that are typically above the editor view of the Eclipse IDE were removed and the IDE only had a button that said "Run/Proceed" that would compile the project and run the automatic tests against the last saved version of the program to see if it was complete. To make the compilation as flexible as possible for different occasions, the button triggers a script to run all the tasks needed to run compilation

```
#include <float.h>
#include "task.h"
#include "marketBasket.h"

using namespace std;

/**
 *  Please write a program, which calculates
 *  the sum of the lowest and highest item using
 *  code similar to the sample code.
 *  FLT_MIN is the minimum value for a float.
 *  FLT_MAX is the maximum value for a float.
 **/
float getLowestHighestSum(marketBasket mb) {
    float highest = FLT_MIN;
    float lowest = FLT_MAX;
    // Implement solution here
    // --------
    function<void (item)> myFunc = [&] (item i){
        if(i.price < lowest){
            lowest = i.price;
        }
        if(i.price > highest){
            highest = i.price;
        }
    };
    mb.iterateOverItems(myFunc);
    // --------
    return highest + lowest;
}
```

Code Sample 10: Task 2 lambda group.

and testing. The script invokes the clang compiler with C++11 arguments to compile the sources and then uses Googletest test-cases to test the programs of the participants. The output of the compilation and the test-cases was saved to a file. This output was printed from the file to the console window at the bottom of the IDE, as is typical. The menus of the IDE were also limited in functionality.

This limited IDE then was extended to save logs of the programmers' behavior while it was running, so that we could measure programmer behavior and the time it took the participants to complete a task. The IDE wrote a log entry every time it was started and when a task was completed successfully, which was triggered by the automatic tests passing. These log messages were written into a hidden file in the project folder and included time-stamps, so that the difference of the time-stamps could be used to calculate the time to completion.

The IDE also logged events for when participants switched between viewing different files. This was done to see if the switching between files was correlated with overall time to completion. Additionally, the IDE created zip-archived snapshots of the project environment every time the participants would try to run their

```
#include <float.h>
#include "task.h"
#include "marketBasket.h"

using namespace std;

/**
 *  Please write a program that returns all the items in
 *  marketBasket with a price under 30 in a vector<item> using code
 *  similar to the sample code.
 **/
vector<item> getAllUnder30(marketBasket mb) {
    vector<item> under30;
    // Implement solution here
    // --------
    marketBasket::iterator marketIter = mb.begin();
    while(marketIter.hasNext()){
        if (marketIter.get().price < 30)
            under30.push_back(marketIter.get());
        marketIter.next();
    }
    // --------
    return under30;
}
```

Code Sample 11: Task 3 iterator group.

code. This enabled us to later extract information about their progress over the span of the experiment. These zip-archives were named with a time-stamp of the time of creation and also contained the output of the last compilation.

Additionally, we also used the software `recordmydesktop` to record a video of the participants' screens for each task. This way we were able to refer back to these videos if we found any problems with the data.

```
#include <float.h>
#include "task.h"
#include "marketBasket.h"

using namespace std;

/**
 *  Please write a program that returns all the items in
 *  marketBasket with a price under 30 in a vector<item> using code
 *  similar to the sample code.
 **/
vector<item> getAllUnder30(marketBasket mb) {
    vector<item> under30;
    // Implement solution here
    // --------
    function<void (item)> find = [&] (item i){
        if(i.price < 30)
          under30.push_back(i);
    };
    mb.iterateOverItems(find);
    // --------
    return under30;
}
```

Code Sample 12: Task 3 lambda group.

Chapter 5

Results

To investigate if the hypotheses are true, the data collected in the experiment were evaluated using descriptive and inference statistics.

5.1 Demographics

		Age		Programming Experience		C++ Experience	
Level of Education	N	Mean	SD	Mean	SD	Mean	SD
Freshman	10	23.60	6.95	2.05	1.55	0.85	0.24
Sophomore	8	23.00	6.57	2.31	1.16	1.69	0.70
Junior	17	24.06	3.73	3.28	1.66	2.44	0.75
Senior	7	27.71	5.50	5.57	3.36	4.29	2.29
Professional	12	31.25	3.70	14.00	5.80	3.42	4.85
total	54	25.89	5.88	5.59	5.60	2.49	2.64

Table 5.1: Age and experience of participants by level of education.

Of the 58 participants that participated in the experiment, 11 identified themselves as female (18.9%). 4 results had to be excluded from the analysis of the results. Of these, three were not following the protocol as instructed and therefore did not produce any usable data. One participant's data was deleted by accident while being transferred from the experiment environment. Of the usable participants, 42 were students of the University of Nevada, Las Vegas, while the 12 remaining were professional developers. The students were divided as follows: 10 were freshmen, 8 sophomores, 17 juniors and 7 seniors, according to our classification (see 4.2). The questionnaire revealed an average age of 25.89 years ($SD = 5.88$) overall. The freshmen had an average age of 23.60 years ($SD = 6.95$), while the sophomores had an average age of 23.00 years ($SD = 6.57$) and the juniors had an average age of 24.06 years ($SD = 3.73$). The Seniors had an average age of 27.71 years ($SD = 5.50$) and the professionals had an average age of 31.25 ($SD = 3.70$).

The participants were asked in the questionnaire how much programming experience they have. Overall, the participants had a mean programming experience of 5.59 years ($SD = 5.60$). The freshman participants

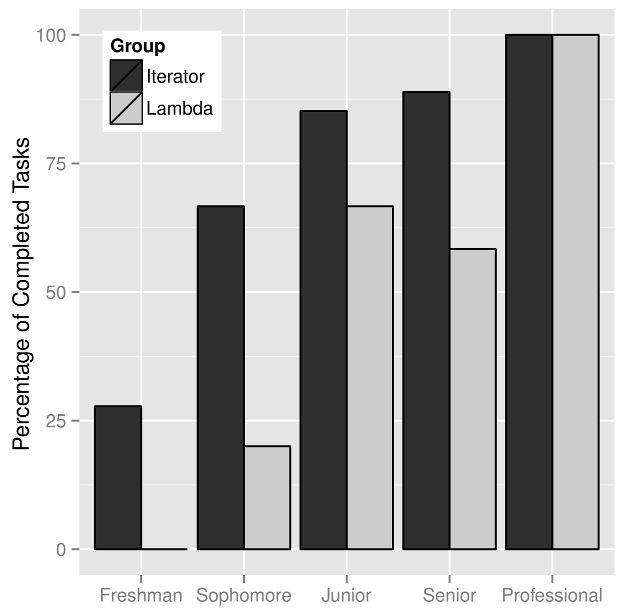

Figure 5.1: The percentage of tasks completed by participants at each experience level.

reported a mean experience of 2.05 years ($SD = 1.55$), the sophomores reported 2.31 years ($SD = 1.16$). The junior participants reported to have 3.28 years of experience on average ($SD = 1.66$) and the seniors average of the reported experience was 5.57 years ($SD = 3.36$). The highest reported programming experience was by the professionals with a mean of 14.00 years of experience ($SD = 5.80$).

Additional to the general programming experience with any kind of language, the questionnaire explicitly asked about the experience with C++ as that was the language being tested. The overall average of experience with C++ was 2.49 years ($SD = 2.64$). The mean experience among freshmen was 0.85 years ($SD = 0.24$), the sophomores had an average experience of 1.69 years ($SD = 0.70$). The juniors reported average experience of 2.44 years ($SD = 2.44$) and the seniors reported a mean of 4.26 years ($SD = 2.29$). Finally, the professionals reported a mean experience of 3.42 years ($SD = 4.85$). An overview of these data can be found in table 5.1.

5.2 Completion of Tasks

The first variable under consideration is the *number of successfully completed tasks.* While the design of the experiment expected all participants to complete the tasks, not all were able to do so. This is especially pronounced in the lambda group as can be seen in figure 5.1. In the graph, it can be seen that, while the freshmen in the experiment could at least sometimes solve the tasks in the iterator group, not one of the participants in the lambda group was able to solve a single task. The issue is not as severe with the higher levels of education in the study, as can be seen when looking at the sophomores, juniors, and seniors. These

			Task 1			
Education	Group	N	min	max	mean	SD
Freshman	Iterator	6	284	2400	1914.83	859.90
	Lambda	4	2400	2400	2400.00	0.00
Sophomore	Iterator	3	1285	2400	1714.67	599.86
	Lambda	5	1477	2400	2215.40	412.78
Junior	Iterator	9	346	2400	1227.89	748.77
	Lambda	8	1643	2400	2069.50	337.58
Senior	Iterator	3	532	767	657.00	125.53
	Lambda	4	2277	2400	2369.25	61.50
Prof.	Iterator	6	169	308	230.33	50.92
	Lambda	6	255	1672	873.83	549.35
Total	Iterator	26	169	2400	1151.52	859.76
	Lambda	27	255	2400	1924.19	676.81

Table 5.2: Solving time of task 1 by group, level and in total.

had a difference in how many tasks were successfully solved, always in favor of the iterator group, but did not fail completely. The sophomores show a large difference between the iterator and the lambda group. Among professionals, all tasks were solved completely within the time limit.

To test these results for significance, an ANOVA was used. The dependent variable in this test was if a task had been completed. The ANOVA was tested with two independent variables. The first was the *level of education*, which was significant, $F(4, 152) = 25.414, p < .001, \eta_p^2 = .401$, and the second was the *group*, which was also significant with $F(1, 152) = 16.094, p < .001, \eta_p^2 = .096$. This shows that $H0_1$ can be rejected.

			Task 2			
Education	Group	N	min	max	mean	SD
Freshman	Iterator	6	515	2400	1777.00	965.20
	Lambda	4	2400	2400	2400.00	0.00
Sophomore	Iterator	3	287	1032	642.67	373.64
	Lambda	5	487	2400	2017.40	855.52
Junior	Iterator	9	274	1054	494.00	237.55
	Lambda	8	283	2400	1396.25	1062.21
Senior	Iterator	3	302	2400	1037.67	1181.07
	Lambda	4	307	2400	942.00	979.39
Prof.	Iterator	6	90	424	263.50	106.93
	Lambda	6	193	589	295.67	146.14
Total	Iterator	26	90	2400	804.81	803.26
	Lambda	27	193	2400	1348.11	1034.03

Table 5.3: Solving time of task 2 by group, level and in total.

5.3 Time to completion

The second variable being analyzed is the *time* it took participants to solve the different tasks to be accepted by the test-cases. This analysis has three independent variables. The first of these is the *task*, from task T1 to task T3, the second is the *group* the participants were in, lambda or iterator, and the last is their *level of education*, broken down into freshman, sophomore, junior, senior an professional.

There was one more participant in the lambda group, with 27, than in the iterator group. All *time* measures are in seconds. The participants of the iterator group took ($M = 1047.56, SD = 887.29$) to solve the tasks. The average for the lambda group, however, was ($M = 1,503.38, SD = 977.70$). Subsampled by the *level of education*, the freshmen took longer to solve the tasks in general with lower times for iterators ($M = 1,981.78, SD = 756.17$) and higher times for lambdas ($M = 2,400, SD = 0.00$). As mentioned in section 5.2 and visible in figure 5.1, all freshmen failed to finish any of the lambda tasks, which leads to every single freshman having the maximum time allowed for the lambda tasks.

The sophomores had shorter times than the freshmen on iterators ($M = 1,358.89, SD = 872.29$) as well as on lambdas ($M = 2,102.93, SD = 646.53$). The juniors' performance on iterators was ($M = 970.48, SD = 735.25$) and ($M = 1,524.63, SD = 943.49$) on lambdas. The seniors were quicker with ($M = 720.56, SD = 660.77$) for iterators and ($M = 1,379.08, SD = 1049.82$) for lambdas. The professionals had the fastest time on iterators ($M = 236,78, SD = 71.13$) as well as on lambdas ($M = 461.39, SD = 438.04$). The comparison of times can be seen in figure 5.2. Further details about the times can be seen in tables 5.2, 5.3, 5.4, and 5.5.

			Task 3			
Education	Group	N	min	max	mean	SD
Freshman	Iterator	6	1521	2400	2253.50	358.85
	Lambda	4	2400	2400	2400.00	0.00
Sophomore	Iterator	3	358	2400	1719.33	1178.95
	Lambda	5	765	2400	2073.00	731.19
Junior	Iterator	9	155	2400	1189.56	869.43
	Lambda	8	187	2400	1108.13	1061.24
Senior	Iterator	3	168	699	449.00	266.85
	Lambda	4	212	2400	826.00	1052.74
Prof.	Iterator	6	147	268	216.50	47.53
	Lambda	6	99	397	214.67	138.20
Total	Iterator	26	147	2400	1186.33	972.40
	Lambda	27	99	2400	1237.85	1063.86

Table 5.4: Solving time of task 3 by group, level and in total.

In an ANCOVA analysis with the fixed factors *group* and *experience* and covariate *task*—to adjust for learning effects—, the results show that the task order was significant and accounted for with $F(2, 159) = 5.985, p = .016, \eta_p^2 = .038$. This can also be seen in figure 5.2. The results for the *group* variable also show statistical significance with $F(1, 160) = 20.123, p < .001, \eta_p^2 = .118$. This shows that about 11.8% of the variation of task times between the participants was accounted for by the difference between iterators and lambdas. This means that $H0_2$ should be rejected as the difference in time between the groups is statistically significant.

When using the variable *experience* the test also shows statistical significance with $F(3, 158) = 31.710, p < .001, \eta_p^2 = .457$. This means that 45.7% of the variance was explained by the difference in experience. This

			Average	
Education	Group	N	mean	SD
Freshman	Iterator	6	1981.78	756.17
	Lambda	4	2400.00	0.00
Sophomore	Iterator	3	1358.89	872.29
	Lambda	5	2101.93	646.53
Junior	Iterator	9	970.48	735.25
	Lambda	8	1524.63	943.49
Senior	Iterator	3	720.56	660.77
	Lambda	4	1379.08	1049.82
Prof.	Iterator	6	236.78	72.13
	Lambda	6	461.39	438.04
Total	Iterator	26	1047.56	887.29
	Lambda	27	1503.38	977.70

Table 5.5: Total average solving time by group, level and in total.

indicates that $H0_5$ can be rejected, as the difference in productivity (*time*) is statistically significant between the levels of experience.

Different ways to analyze the data were tested but came to the same broad conclusions. Thus, we chose a conservative statistical model.

5.4 Errors

The last dependent variables under investigation were the *number of errors* the participants made while programming and how much *time* was *spent while fixing errors* in relation to the total time to solve the task. The errors were extracted from the participants' compilation events that were recorded during the experiment as described in 4.5. The compilation output was investigated for error messages and the amount of errors per compilation output was established as the number of distinct errors in the output. This means, that there was a binary variable for each error message that appeared in all of the compilation outputs that indicated if a specific error exists in an error message. The number of errors in a compilation event is the sum of these variables. This way, if an error appears twice or more times in a compilation event, it is only counted once. The *amount of time spent solving errors* is calculated by taking the time-stamps that are connected to the compilation events and taking the time differences between the first compilation event in a sequence of non-compiling events and the first compilation event after this sequence started in which the program compiles again. These time-intervals are then added together for the total amount of time spent fixing non-compilable states.

The descriptive statistics for the number of errors encountered show that, generally, the lambda group had more errors ($M = 17.77, SD = 28.58$) than the iterator group ($M = 8.75, SD = 12.32$). This is also observable when the amount of errors is broken down by task. Task 1 has ($M = 27.89, SD = 27.38$), 2 has ($M = 10.16, SD = 17.16$) and 3 has ($M = 13.55, SD = 36.78$) for group lambda. The iterator group has lower

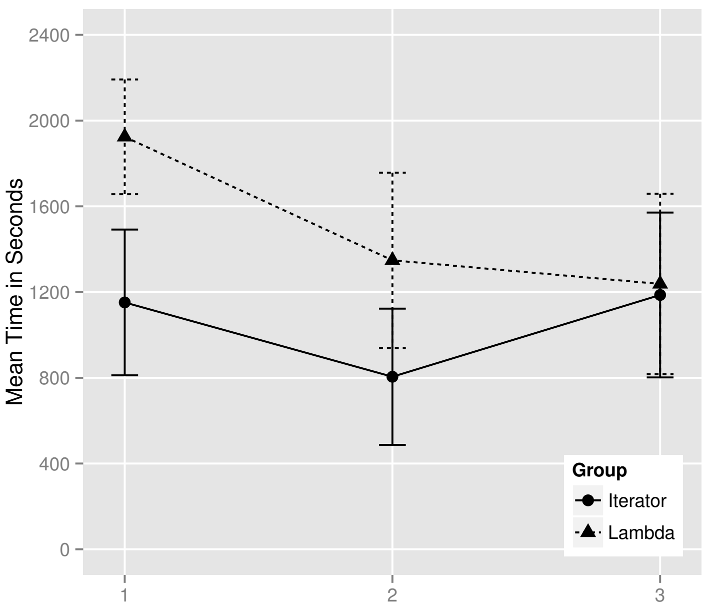

Figure 5.2: Group times compared.

	T1		T2		T3		total	
group	mean	SD	mean	SD	mean	SD	mean	SD
lambda	27.89	27.38	10.16	17.16	13.55	36.78	17.77	28.58
iterator	11.23	13.98	5.08	9.15	9.78	12.81	8.75	12.32

Table 5.6: Number of errors.

means with ($M = 11.23, SD = 13.98$) for task 1, ($M = 5.08, SD = 9.15$) for 2 and ($M = 9.78, SD = 12.81$) for task 3.

An ANOVA shows that there is a significant difference between the groups with $F(1, 148) = 5.704, p = .018, \eta_p^2 = .039$ and for the tasks, $F(2, 148) = 4.093, p = .019, \eta_p^2 = .055$. This shows that 3.9% of the variance in errors can be explained by the groups the participants were in and 5.5% of the variance can be explained by the differences in the tasks. This means we can reject $H0_3$.

	T1		T2		T3		total	
group	mean	SD	mean	SD	mean	SD	mean	SD
lambda	65.05	34.32	45.43	37.34	57.75	33.13	56.37	35.54
iterator	57.03	31.34	26.14	24.88	48.54	31.88	44.20	31.99

Table 5.7: Percent of time spent fixing errors.

Looking at the amount of *time spent on fixing errors*, the pattern was similar. The mean percentage time spent on working on fixing errors for the lambda group was ($M = 56.37, SD = 35.54$), while the iterator group's mean was ($M = 44.20, SD = 31.99$). The average for task 1 for the lambda group was

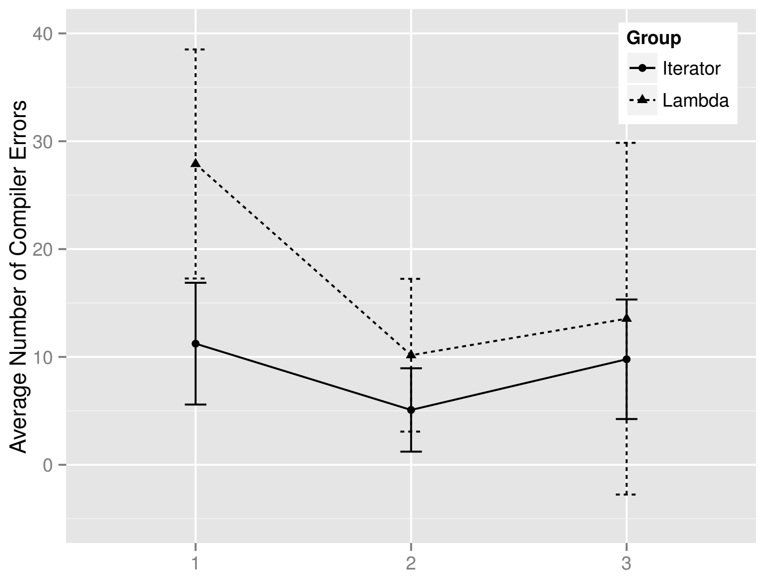

Figure 5.3: Compiler errors compared.

$(M = 65.05, SD = 34.32)$ compared to $(M = 57.03, SD = 31.34)$ average for the iterator group. For the second task, $(M = 45.43, SD = 37.34)$ was observed for the lambda group and $(M = 26.14, SD = 24.88)$ for the iterator group. In task 3, the amount of time spent on fixing errors went up for both the groups compared to their second tasks with an average of $(M = 57.75, SD = 33.13)$ for the lambda group and $(M = 48.54, SD = 31.88)$ for the iterator group. An ANOVA shows that there is a significant difference between the groups, $F(1, 148) = 5.178, p = .024, \eta_p^2 = .035$. Additionally, there is a significant difference between the tasks, $F(2, 148) = 8.027, p < .001, \eta_p^2 = .102$, thus hypothesis $H0_4$ should be rejected as well.

Chapter 6

Discussion

As can be seen in the results, the null hypotheses can be rejected based on the analysis of the data. Especially less trained programmers seemed to struggle with the programming tasks, as can be seen in the number of completed tasks for freshmen and sophomores. The general results show that there is a significant difference between how many tasks were solved between the lambda and iterator group. The professionals in the experiment do, however, not show any difference in how many tasks were solved between the groups. Investigating $H0_5$ also shows that professional and student performance might not be comparable. With this information, one could make the argument that the difference between the number of solved tasks between the groups only holds true for students and not professionals.

This distinction does not hold for the amount of time spent to solve tasks, as the times indicate that the professionals in the lambda group took significantly longer to solve the tasks than the iterator group, as do all the student levels, too. The timing results do, however, show a decline over the course of the experiment. It seems as if participants got used to the features they were working with and the style of task they were given over the time of the experiment. For example, sophomores and professionals reach lower mean times for the lambda group than the iterator group in task 3, with a higher standard deviation. This seems to indicate that programmers might be able to reach parity performance with the lambda function feature in relatively short time. The experiment took the participants only about one and a half hours to complete.

On the one hand, speculatively, if parity performance persists after a programmer spent about 1.5 hours on the new feature, then introduction of the feature would still be a time investment with no return. If one assumes that every programmer would have to spent 1.5 hours to learn the feature without any real gain, then this would be a net loss. That is, totally ignoring the cost of designing and implementing the feature into the language and the change in textbooks that would be necessary.

On the other hand, it is not predictable if the performance for programmers using lambdas would increase further and times to solve problems in the future would decrease to a level in which the initial time-commitment to learn the feature is made up over time and there will be a net gain to the industry using the language, especially if the accumulated productivity can offset the initial implementation and textbook changes.

Both of these arguments fully ignore the fact that lambda functions can have more than this one use, of course. A recommendation to language designers based on the presented data can only be made hesitantly as there is more research to be done on the effect of lambda functions (see chapter 8). Currently, implementation of lambda functions into object-oriented lambda functions should be seen critically, as no benefit has been shown so far.

Compiler error results shown in this experiment indicate that there is a significant difference in how many errors are encountered by the participants between the two groups and how much time was spent fixing the errors. A possible reason for this could be the syntax and error design of C++.

It could be argued that C++'s syntax choice for lambdas is not optimal if the results of [SS13] are taken into account. Apart from the generally bad results for C-like syntax in the publication, as exemplified by Java, the paper also shows bad results for parentheses, brackets and braces with novices. All of which are heavily featured in the syntax design of C++ lambdas. Additionally, the type for functions in C++ uses a combination of angle brackets and parentheses without an explicit explanation of their meaning. An example of this is `function<float (int, int)>`. This use of templating-style types, with the distinction between the return type `float` and the parameter types `(int, int)` is, to the author's knowledge, not used anywhere in a similar way and might confuse programmers with knowledge of templating in C++, as well as programmers which are mostly unaware of the concept to begin with.

It should be mentioned, however, that this argument does make a logical leap, in that it equates the results found in novices, which were completely inexperienced in programming languages, with a wide range of proficiencies. All of the participants had at least had a semester's worth of C++ programming experience. It may be that participants found the syntax not challenging.

On the topic of compiler errors for C++ lambdas, the messages might just not be very helpful for programmers. Studies have shown that type errors could be, next to unresolved identifiers and forgotten semicolons, a hard type of compiler error to solve [DLRT12, AB15]. While a formal investigation of the errors encountered in this experiment is out of scope of this publication, a short analysis of the errors finds that the third most occurring[1] error in the experiment was that there was no viable conversion between two types[2]. One can of course argue about the quality of the error messages, but, qualitatively, clang's error messages do not seem to be particularly informative. One example would be the error "error: member reference base type "int" is not a structure or union", which seems hard to understand.

The analysis shows a significant difference between the performance of students and professionals. The often cited[3] Höst et al. [HRW00] states that their findings will probably hold true when comparing graduate

[1]As found in the collection of compilation outputs. Note that errors can overlap in the error outputs.

[2]The first most often occurring problem is "member not found" and the second is "use of undeclared identifier", which matches well with the findings of the cited study.

[3]428 times by the time of this writing on April 6th 2016, according to Google Scholar.

students and professional developers. Since the students used in this experiment are undergraduates, a comparison with this study is less feasible. Especially when taking into account that Höst et al. focus their experiment on software engineering based rating tasks instead of programming, which is the focus of this experiment. Salman et al. [SMJ15] focus on programming tasks and find no significant difference between students and professionals when they are working with previously unfamiliar skills. Their experiment does find a difference for the use of familiar skills. As stated earlier, there seems to be a learning effect present for the participants of the experiment, which suggests, that participants were unfamiliar with the programming language features used in the experiment. Then the results this experiment contradict the findings of Salman et al. . The findings hold up well when compared to Wiedenbeck [Wie85] and Youngs [You74] in that the professionals seemed to be faster and have fewer errors than the students.

Chapter 7

Limitations and Threats to Validity

While a considerable amount of time was spent trying to design an experiment to formally evaluate this issue, a single experiment is always limited. This chapter will discuss the validity of this study's design and generalizability.

7.1 Internal Validity

Learning Effect: The experiment was designed in a way so that the tasks were always solved in the same order. One could argue that the complexity of the tasks does not change from task to task and that the actions required of the students might be repetitive as the steps to solve each task are very similar to solving the previous task and only slight changes to the code have to be made. This has resulted in a learning effect as was shown in the analysis in section 5.3. While the effect is taken into account as a covariate in the results, it might have been better to design the experiment in a way that reduces learning effects by changing the context of the experiment more thoroughly between tasks.

Time constraint: Due to limitations on recruitment, ethics, and practicality, the time each student could spent to solve a task was limited to 40 minutes. Participants are only willing to participate if they can fit the experiment into their busy schedule and 1-2% of extra credit are only worth a certain amount of effort for students. Additionally, it should be considered how much time commitment can be ethically expected from participants. So, when the cut-off point was reached, the participants were asked to move on to the next task, regardless of how close they were to solving the task or not. This results in a ceiling for the timing and might group participants who were about to solve the task with participants who were far from reaching the right solution. If participants had more time to solve the tasks, they might have learned more about the construct they were using in their first task and then moved on to solve the other two tasks quickly, which could have influenced the studies results.

Recruitment Bias: It is noteworthy, that the students involved in the experiment were recruited from computer science classes and rewarded with extra credit towards their grades for that class. This might have encouraged especially under-performing students to participate in the experiment, which might have effected the outcome of the study. In contrast, the professional participants that were recruited using a mailing list and Twitter. It is likely that these participants were highly motivated programmers as they seem to be engaging with social media related to programming experiments in their free time and accepted considerable inconvenience and time commitment to participate in an experiment which would not benefit them directly. As such, they might not represent the population of all professional programmers accurately. This might have effected how well these developers performed compared to possibly under-performing students.

7.2 External Validity

Limit of Scope: The tasks were designed in a way that the difference between iterators and lambda functions could be measured in respect to productivity. The tasks therefore completely focus on iteration on data structures as iterators are exclusively designed to fulfill this task. This limits the experiment to only be able to provide information about the iteration aspect of lambda functions, leaving out other aspects such as parallelization. This is due to the limited nature of randomized controlled trials, as they only have a limited time-frame. Future studies of a similar nature should investigate the other aspects of lambda functions to gain a more complete understanding of their impact on development.

Previous Knowledge of Participants: The student-participants that were recruited for this experiment were all students at the University of Nevada, Las Vegas. The findings might not be generalizable to students of other institutions as different teaching methodologies or material might change results. On the other hand, selecting individuals from different levels of the educational pipeline, before and after they learned about specific elements of the C++ programming language, might have prevented any effects that possible differences between institutions might have made.

Number of Participants: The number of participants for this study was limited to 54 individuals. This has resulted in statistical significant results, but it is still easy to argue that this is a very limited number of participants and not a representative sample of the population. This could limit the generalizability of the study.

Choice of Language: This study only focused on the C++ programming language, therefore, it is likely that the results of this study will not hold true for every other programming language. It seems possible that the syntax choices made when designing lambda functions in C++ might partly explain the outcome of this study. Like discussed in chapter 3, syntax choices have been shown to matter at least for programming novices.

Simplicity of the Tasks: The tasks were designed to be simple on purpose. This has to do with the limited time frame of the experiment as well as with trying to limit the impact on other complicating factors on the times measured for the experiment. The design is chosen in a way that should prevent participants from having to think much about other parts of the programming language than the ones important to the topic of the study. This is why it was chosen to have participants only fill in a specific part of a method as well as ignore the possible different capture list options by just using one approach. This might limit generalizability in the sense that programmers in a real-world setting might have to write extra code, like the iterator itself or the method which applies the lambda function, which might have different effects on the programmers productivity than measured in this specific experiment. Other effects might come into play from differences in general architectural design, which might impact the usefulness of either of the programming language constructs.

Chapter 8

Future Work

As discussed in chapter 7, the study was not completely generalizable to all aspects of lambda functions. Thus, apart from necessary replications of the presented study, research needs to also focus on other use-cases of lambda functions and if debugging lambda based code is harder than other code. These might include the use in parallel streams as were introduced into Java 8 or listener functions for event driven programming which can be found in UI programming or server applications. C++ is also not the only object-oriented programming language to recently introduce this feature and as such, replication studies in other language like the aforementioned Java 8 or C# might give valuable information about the usability and usefulness of lambda functions in different languages. Such research could also help to find out more about the impact different approaches to syntax choices for lambda functions might make on developer productivity.

If future studies find that there is indeed a benefit to having lambda functions in broadly-used object-oriented programming languages, further experiments should investigate the details for the best syntax choices. This could, for example, involve the use of placebo languages, in which syntax elements are chosen at random, as control groups similar to [SS13]. Initial ideas about possible syntax choices might be gained from analyzing what kind of errors programmers make while using lambdas in languages like Java since the introduction of the new feature.

Another interesting research direction for lambda functions could be to investigate existing usage patterns in the languages that have recently introduced the lambda function feature. Questions to ask in this regard might be about how well developers adopted the feature, as in how often it is used instead of other features, how often it is used in total, which kind of project adopted their use, whether the feature was deliberately used to replace existing code or if it is mostly used for new code, and which errors developers often encounter while using the new feature. Some of these questions might be answered by investigating source code repositories, others could be investigated by surveys.

Other interesting insights might come from longer term studies on the use of lambda functions in a controlled setting. This could be done by giving the participant a longer term project to work on in multiple sessions. Such an experiment might make it possible to make the findings more generalizable as they would resemble actual programming work more closely.

Apart from further research into lambda functions, other programming language constructs need further investigation. Such a feature might be enumerated types, which exist in programming languages such as Java.

Chapter 9

Conclusion

This thesis described the design and execution of an experiment to gain insight into how lambda functions effect developer productivity. The experiment is part of a larger set of experiments to gain knowledge on how aspects of programming languages effect developers to help the design of future languages. The experiment had 54 participants which were recruited at the University of Nevada, Las Vegas and on the internet. The experience of the participants varied widely, which enabled the study to also give insight into a common claim that the results of experiments with only student participants are generalizable to professional developers.

None of the freshman students in the study were able to complete the lambda-based tasks in the given time frame, while the group solving the iterator tasks solved about a quarter of their tasks successfully. Percent of completion for both groups increased with experience and only professional developers solved task at equal rates between the groups.

Statistically, the results of the study show that there was a significant negative impact on the productivity of participants when they used lambda functions as opposed to iterators for iteration tasks. This finding might be explained by a learning effect as the difference between the times to completion between the groups shrinks with the ongoing experiment.

It was also found that the lambda group produced significantly more compiler errors while working on the tasks than the iterator group. Furthermore, participants of the lambda group spent more time fixing errors than participants of the iterator group.

Analysis of the differences between experienced and less experienced developers shows that experienced developers performed significantly better than their peers in how many tasks were completed and how much time was needed to complete programming tasks correctly. This casts doubt on the claims that student performance can be generalized to the entire population of developers.

As this is only a first study to investigate the usefulness of lambda functions for procedural or object-oriented programming languages, no recommendations can be made as of yet whether to implement them. More research has to be done to validate the findings and investigates possible benefits for other use-cases.

Bibliography

[AB15] Amjad Altadmri and Neil C.C. Brown. 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 522–527, New York, NY, USA, 2015. ACM.

[ATCL$^+$98] Ali-Reza Adl-Tabatabai, Michał Cierniak, Guei-Yuan Lueh, Vishesh M Parikh, and James M Stichnoth. Fast, effective code generation in a just-in-time java compiler. In *ACM SIGPlAN Notices*, volume 33, pages 280–290. ACM, 1998.

[Ben13] Timothy Beneke. The javaone 2013 technical keynote. `https://blogs.oracle.com/javaone/entry/the_javaone_2013_technical_keynote`, 2013. [Online; accessed 24-March-2015].

[Boe88] Barry W Boehm. Understanding and controlling software costs. *Journal of Parametrics*, 8(1):32–68, 1988.

[BS96] David F Bacon and Peter F Sweeney. Fast static analysis of c++ virtual function calls. *ACM Sigplan Notices*, 31(10):324–341, 1996.

[Car98] Michelle Cartwright. An empirical view of inheritance. *Information and Software Technology*, 40(14):795–799, 1998.

[CC05] Karen M Conrad and Kendon J Conrad. Compensatory rivalry. *Encyclopedia of Statistics in Behavioral Science*, 2005.

[Chu32] Alonzo Church. A set of postulates for the foundation of logic. *Annals of mathematics*, 2(33):346–366, 1932.

[CKS15] Igor Crk, Timothy Kluthe, and Andreas Stefik. Understanding programming expertise: An empirical study of phasic brain wave changes. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 23(1):2, 2015.

[DBM$^+$96] J. Daly, A. Brooks, J. Miller, M. Roper, and M. Wood. Evaluating inheritance depth on the maintainability of object-oriented software. *Empirical Software Engineering*, 1(2):109–132, 1996.

[DLRT12] Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. All syntax errors are not equal. In *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education*, ITiCSE '12, pages 75–80, New York, NY, USA, 2012. ACM.

[DLRTH11] Paul Denny, Andrew Luxton-Reilly, Ewan Tempero, and Jacob Hendrickx. Understanding the syntax barrier for novices. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*, ITiCSE '11, pages 208–212, New York, NY, USA, 2011. ACM.

[EHRS14] Stefan Endrikat, Stefan Hanenberg, Romain Robbes, and Andreas Stefik. How do api documentation and static typing affect api usability? In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 632–642. ACM, 2014.

[FBM^{+}14] Thomas Fritz, Andrew Begel, Sebastian C Müller, Serap Yigit-Elliott, and Manuela Züger. Using psycho-physiological measures to assess task difficulty in software development. In *Proceedings of the 36th International Conference on Software Engineering*, pages 402–413. ACM, 2014.

[FKR^{+}00] Robert Fitzgerald, Todd B Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: An optimizing compiler for java. *Software-Practice and Experience*, 30(3):199–232, 2000.

[Gan77] J. D. Gannon. An experimental evaluation of data type conventions. *Commun. ACM*, 20(8):584–595, 1977.

[Han10] Stefan Hanenberg. Faith, hope, and love: an essay on software science's neglect of human factors. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 933–946, New York, NY, 2010. ACM.

[HCIB04] T Dean Hendrix, James H Cross II, and Larry A Barowski. An extensible framework for providing dynamic data structure visualizations in a lightweight ide. In *ACM SIGCSE Bulletin*, volume 36, pages 387–391. ACM, 2004.

[HH13] Michael Hoppe and Stefan Hanenberg. Do developers benefit from generic types?: An empirical comparison of generic and raw types in java. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 457–474, New York, NY, USA, 2013. ACM.

[HM10] Brian Harvey and Jens Mönig. Bringing "no ceiling" to scratch: can one language serve kids and computer scientists. *Proc. Constructionism*, 2010.

[HRW00] Martin Höst, Björn Regnell, and Claes Wohlin. Using students as subjects-a comparative study of students and professionals in lead-time impact assessment. *Empirical Software Engineering*, 5(3):201–214, 2000.

[JFC08] Jaakko Järvi, John Freeman, and Lawrence Crowl. Lambda expressions and closures: Wording for monomorphic lambdas (revision 4). Technical report, Tech. Rep, 2008.

[Kai15] Antti-Juhani Kaijanaho. The extent of empirical evidence that could inform evidence-based design of programming languages: A systematic mapping study. *Jyväskylä Licentiate Theses in Computing, University of Jyväskylä*, 2015.

[KHR^{+}12] Sebastian Kleinschmager, Stefan Hanenberg, Romain Robbes, Éric Tanter, and Andreas Stefik. Do static type systems improve the maintainability of software systems? An empirical study. In *IEEE 20th International Conference on Program Comprehension, ICPC 2012, Passau, Germany, June 11-13*, pages 153–162, 2012.

[KM09] Andrew J Ko and Brad A Myers. Finding causes of program output with the java whyline. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1569–1578. ACM, 2009.

[KPBB^{+}09] Barbara Kitchenham, O. Pearl Brereton, David Budgen, Mark Turner, John Bailey, and Stephen Linkman. Systematic literature reviews in software engineering - a systematic literature review. *Inf. Softw. Technol.*, 51(1):7–15, January 2009.

[LFH10] Jonathan Lazar, Jinjuan Heidi Feng, and Harry Hochheiser. *Research methods in human-computer interaction*. John Wiley & Sons, 2010.

[NF15] Sebastian Nanz and Carlo A Furia. A comparative study of programming languages in rosetta code. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 778–788. IEEE Press, 2015.

[NM08] Austin Lee Nichols and Jon K Maner. The good-subject effect: Investigating participant demand characteristics. *The Journal of general psychology*, 135(2):151–166, 2008.

[PBMH11] Chris Parnin, Christian Bird, and Emerson R. Murphy-Hill. Java generics adoption: how new features are introduced, championed, or ignored. In *Proceedings of the 8th International Working Conference on Mining Software Repositories, MSR 2011 (Co-located with ICSE), Waikiki, Honolulu, HI, USA, May 21-28, 2011, Proceedings*, pages 3–12. IEEE, 2011.

[PHR14] Pujan Petersen, Stefan Hanenberg, and Romain Robbes. An empirical comparison of static and dynamic type systems on api usage in the presence of an ide: Java vs. groovy with eclipse. In *22nd International Conference on Program Comprehension, ICPC 2014, Hyderabad, India, June 2-3, 2014*, pages 212–222. ACM, 2014.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[PSV15] Andrew Petersen, Jaime Spacco, and Arto Vihavainen. An exploration of error quotient in multiple contexts. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, pages 77–86. ACM, 2015.

[PUPT03] Lutz Prechelt, Barbara Unger, Michael Philippsen, and Walter Tichy. A controlled experiment on inheritance depth as a cost factor for code maintenance. *Journal of Systems and Software*, 65(2):115–126, 2003.

[RHW10] Christopher J Rossbach, Owen S Hofmann, and Emmett Witchel. Is transactional programming actually easier? *ACM Sigplan Notices*, 45(5):47–56, 2010.

[RPFD14] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 155–165. ACM, 2014.

[Sam06] Valentin Samko. A proposal to add lambda functions to the c++ standard. Technical report, Technical Report–N1958=06-0028, 2006.

[SF14] Carlos Souza and Eduardo Figueiredo. How do programmers use optional typing?: an empirical study. In *Proceedings of the 13th international conference on Modularity*, pages 109–120. ACM, 2014.

[SH14] Samuel Spiza and Stefan Hanenberg. Type names without static type checking already improve the usability of apis (as long as the type names are correct): an empirical study. In *13th International Conference on Modularity, MODULARITY '14, Lugano, Switzerland, April 22-26, 2014*, pages 99–108. ACM, 2014.

[SHM$^+$14] Andreas Stefik, Stefan Hanenberg, Mark McKenney, Anneliese Amschler Andrews, Srinivas Kalyan Yellanki, and Susanna Siebert. What is the foundation of evidence of human factors decisions in language design? an empirical study on programming language workshops. In *Proceedings of the 2014 IEEE 20th International Conference on Program Comprehension*, ICPC '14, pages 223–231. IEEE Computer Society, 2014.

[SKL$^+$14] Janet Siegmund, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hanenberg. Measuring and modeling programming experience. *Empirical Software Engineering*, 19(5):1299–1334, 2014.

[SMJ15] Iflaah Salman, Ayse Tosun Misirli, and Natalia Juristo. Are students representatives of professionals in software engineering experiments? In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 666–676. IEEE, 2015.

[SS13] Andreas Stefik and Susanna Siebert. An empirical investigation into programming language syntax. *Trans. Comput. Educ.*, 13(4):19:1–19:40, November 2013.

[Str14] Bjarne Stroustrup. C++11 - the new iso c++ standard. `http://www.stroustrup.com/C++11FAQ.html#lambda`, 2014. [Online; accessed 24-March-2015].

[VRCG$^+$99] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot-a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[Wei14] Tal Weiss. The dark side of lambda expressions in java 8. `http://blog.takipi.com/the-dark-side-of-lambda-expressions-in-java-8/`, 2014. [Online; accessed 24-March-2015].

[Wie85] Susan Wiedenbeck. Novice/expert differences in programming skills. *International Journal of Man-Machine Studies*, 23(4):383–390, 1985.

[WJG$^+$06] J. Willcock, J. Järvi, D. Gregor, B. Stroustrup, and A. Lumsdaine. Lambda functions and closures for c++. Technical Report N1968=06-0038, ISO/IEC JTC 1, Information technology, Subcommitee SC 22, Programming Language C++, February 2006.

[You74] Edward A Youngs. Human errors in programming. *International Journal of Man-Machine Studies*, 6(3):361–376, 1974.

[ZBT11] He Zhang, Muhammad Ali Babar, and Paolo Tell. Identifying relevant studies in software engineering. *Inf. Softw. Technol.*, 53(6):625–637, June 2011.

Curriculum Vitae

Graduate College
University of Nevada, Las Vegas

Phillip Merlin Uesbeck

Local Address:
3955 Swenson Street Apt. 14
Las Vegas, Nevada 89119

Degrees:
Bachelor of Science in Applied Computer Science – Software Enigneering 2014
Universität Duisburg-Essen

Publications:
(In Press) Uesbeck, P. M., Stefik, A., Hanenberg, S., Pedersen, J., and Daleiden, P. "*An Empirical Study on the Impact of C++ Lambdas and Programmer Experience*" Software Engineering (ICSE), 2016 IEEE/ACM 38th IEEE International Conference on. Vol. 1. IEEE, 2016.

Thesis Title: An Empirical Study on the Impact of C++ Lambdas and Programmer Experience

Thesis Examination Committee:
Chairperson, Dr. Andreas Stefik, Ph.D.
Committee Member, Dr. Jan "Matt" Pedersen, Ph.D.
Committee Member, Dr. Kazem Taghva, Ph.D.
Graduate Faculty Representative, Dr. Matthew Bernacki, Ph.D.